Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186–5140
Volume 4, Number 1, pages 69–73, January 2015

Accelerating RSA encryption using GPUs

Ryosuke Sakai, Koji Nakano, and Yasuaki Ito
Department of Information Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-hiroshima, 739-8527 Japan

***Abstract*—RSA encryption is one of the most well known algorithms for public-key cryptography. This paper presents a GPU (Graphics Processing Unit) implementation of RSA encryption. In our approach, we used two ideas to accelerate it. The first idea is efficient memory access for GPU architecture. The second idea is to compute the sum of products for multiple length arithmetic operation using inline assembler. The experimental results show that our GPU implementation on NVIDIA GeForce GTX TITAN attains a speed-up factor of 53.2 for 1024-bit RSA encryption and 56.4 for 2048-bit RSA encryption over the single CPU implementation.**

***Index Terms*—RSA encryption, Montgomery modular multiplication, GPU, CUDA**

I. はじめに

RSA 暗号 [1] は，通信等の分野において現在広く使われている公開鍵暗号である．RSA 暗号では，桁数の大きい合成数の素因数分解が困難であることを安全性の根拠としており，安全性を保つためには 1024bit 以上の桁数が必要となる．また，暗号化・復号化の際にべき乗剰余演算を利用するため，計算量が膨大になり，非常に時間がかかってしまう．そこで，GPU(Graphics Processing Unit) を用いて RSA 暗号を実装することにより，計算の高速化を図った．GPU は，グラフィックス処理に特化したハードウェアであり，非常に高い演算能力を持ち，その性能は現在も向上し続けている．このような GPU をグラフィックス処理だけではなく，汎用的な処理の並列処理に利用しようとする技術のことを GPGPU(General-purpose computing on GPU) と呼び，この技術を用いることで様々な処理を並列かつ高速に行うことができる．また，マルチコア CPU 等の同等の並列性を持つ他のデバイスと比較して安価であることから，様々な研究開発が行われている．この GPGPU を実現するために，NVIDIA 社は CUDA[2], [3], [4] と呼ばれる C 言語の統合開発環境を提供している．本研究では CUDA を用いて RSA 暗号を実装し，高速化することを目的としている．

本稿で提案する GPU 実装では，1 つのスレッドに 1 つの平文の暗号化を割り当てる．各スレッドは 1 つの平文を逐次的に暗号化し，複数のスレッドによって大量の平文を並列に暗号化する．高速化を実現するために，効率的なメモリアクセスの実装，インラインアセンブラの利用を行った．これにより，NVIDIA 社の GeForce GTX TITAN に実装した結果，CPU で逐次処理した結果と比較して，1024bit の暗号化で 53.2 倍，2048bit の暗号化で 56.4 倍の高速化を達成した．また，既存手法である Jang らの実装 [5] に対して，8.5 倍のスループットを得た．

II. RSA 暗号

RSA 暗号 [1] は，通信等の分野において現在広く使われている公開鍵暗号であり，暗号化と復号化にはそれぞれ，公開鍵と秘密鍵の別個の鍵を用いる．暗号化に用いる公開鍵は，通信相手に公開し，復号化に用いる秘密鍵は，公開しない．ある平文 P を暗号化するためには，適切に選択した法 M と暗号鍵 E を用いて，暗号文 $C = P^E \bmod M$ を求める．復号化するためには，復号鍵 D を用いて，同様に $P = C^D \bmod M$ を求める．

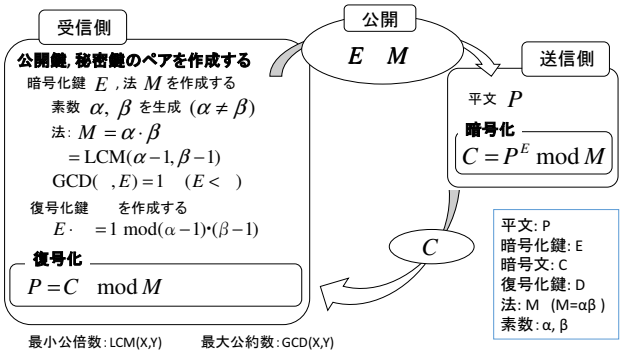


Fig. 1. RSA 暗号の流れ

RSA 暗号の具体的な流れを図 1 に示す．まず，暗号鍵である E と法 M を任意に選択した十分に大きな素数の組 α, β から決定する ($\alpha \neq \beta$)．法 M は $M = \alpha \cdot \beta$ である．暗号鍵 E は，$\alpha - 1, \beta - 1$ の最小公倍数より小さく，かつ互いに素である任意の値が選択される．多くの場合，$E = 65537$ が利用されており，本実装でもそれを採用している．復号鍵 D は $E \cdot D = 1 \bmod (\alpha - 1) \cdot (\beta - 1)$ を満たす正の整数である．RSA 暗号は，桁数が大きい合成数の素因数分解の難しさを安全性の根拠としており，安全性を保つためには 1024bit 以上の桁数が必要となる．また，暗号化・復号化の際にべき乗剰余演算を利用するため，計算量が膨大になり，非常に時間がかかってしまう．そこで，べき乗剰余演算を高速に行うことができるモンゴメリ乗算が利用される．

A. モンゴメリ乗算

モンゴメリ乗算 [6] は，べき乗剰余演算の計算に必要な乗算剰余演算を高速に行うアルゴリズムである．一般に計算機上での剰余演算は，加算や乗算に比べて時間がかかる．そこで $(X + q \cdot M) \equiv X \bmod M$ の性質を利用して，加算後の下位ビットが 0 になるように $q \cdot M$ を加えることで，剰余演算を 2 のべき乗の除算，つまりビットシフトに置換することができる．このため，乗算，加算，ビットシフトのみを用いて高速に乗算剰余演算を計算でき，これを繰り返し計算することでべき乗剰余演算を高速に計算することができる．

以下に，本実装に利用したモンゴメリ乗算アルゴリズム (FIOS method)[7], [8] を示す．それぞれ Rbit の A, B, M と M の逆元である m'_0 を入力として，$A \cdot B \cdot 2^{-dr} \bmod M$ を計算する. ここで，dr は法 M のビット幅であり $dr = R$ である．r は基数 ($r = 32$)，d は分割数を示す．例えば，ビット幅が 2048 のとき，$d = 64$ となる．法 M の逆元である m'_0 は，$m'_0 = -m_0^{-1} \bmod 2^r$ で求める．最終的な P の値は $P < 2M$ の値となるので，$P \geq M$ となった場合は M を減算することで，最終的な解を求める．FIOS 手法を利用することで，必要メモリの減少，乗算剰余演算の高速化を実現できる．

Algorithm 1 Montgomery Multiplication(FIOS method)

Input: A, B, M, m'_0
Output: $P = A \cdot B \cdot 2^{-dr} \bmod M$
1: $P \leftarrow 0$;
2: **for** i **from** 0 **to** $d-1$ **do**
3: $(u, v) \leftarrow a_0 b_i + p_0$;
4: $t \leftarrow u$;
5: $q \leftarrow v m'_0 \bmod 2^r$;
6: $(u, v) \leftarrow m_0 q + v$;
7: **for** j **from** 1 **to** $d-1$ **do**
8: $(u, v) \leftarrow a_j b_i + t + u$;
9: $t \leftarrow u$;
10: $(u, v) \leftarrow m_j q + p_j + v$;
11: $p_{j-1} \leftarrow v$;
12: **end for**
13: $(u, v) \leftarrow p_d + t + u$;
14: $p_{d-1} \leftarrow v$;
15: $p_d \leftarrow u$;
16: **end for**
17: **if** $P \geq N$ **then**
18: **return** $P - N$;
19: **else**
20: **return** P;
21: **end if**

B. バイナリ法を用いたべき乗剰余演算

べき乗剰余演算は乗算剰余演算の繰り返しであるため，バイナリ法を用いることで乗算剰余演算の回数を減らすことができる. バイナリ法と Algorithm 1 のモンゴメリ乗算を組み合わせたべき乗剰余の計算アルゴリズムを Algorithm 2 に示す. それぞれ Rbit の $P, M, 2^{2dr} \bmod M$, $n$bit の E を入力として, $P^E \bmod M$ を計算する. Algorithm 1 と同様に r はモンゴメリ乗算の基数を示し, $dr = R$ である. バイナリ法とはべき乗演算の際に, 指数を 2 進数表現 $E = (E_{n-1}, ..., E_1, E_0)$ で扱い, 各ビット E_i の 0, 1 に対応して底の二乗と乗算を繰り返すことで, 乗算の回数を減らす手法である. まず, 1 行目で 1 と $2^{2dr} \bmod M$, 2 行目で平文 P と $2^{2dr} \bmod M$ のモンゴメリ乗算を行うことでモンゴメリ領域への写像を行っている．モンゴメリ領域への写像とは，アルゴリズム中の P, C を $k \cdot 2^R \bmod M$(k は任意の正の整数) のように，モンゴメリ乗算が実行可能な形式に変換することをいう．3-8 行目ではバイナリ法を用いてべき乗剰余演算を計算している. 指数 E の各ビットの値に対応して乗算と二乗を繰り返すことで, 最悪で $2n$ 回, 平均で $1.5n$ 回のモンゴメリ乗算の実行でべき乗剰余演算を行える. 9 行目で C と 1 のモンゴメリ乗算を行い, モンゴメリ領域から元の形式に戻すことで $P^E \bmod M$ を出力する. モンゴメリ領域への写像に 2 回，モンゴメリ領域から通常の形式に戻すのに 1 回のモンゴメリ乗算が必要になるため，この手法を用いたべき乗剰余演算は最悪の場合に, モンゴメリ乗算を $2n+3$ 回行う. アルゴリズム中の下線はモンゴメリ乗算を行う処理であることを示す.

Algorithm 2 Modulo Expnentiation

Input: $P, E, M, -M^{-1}, 2^{2dr} \bmod M$
Output: $C = P^E \bmod M$
1: $C := \underline{1 \cdot (2^{2dr} \bmod M) \cdot 2^{-dr} \bmod M}$
2: $P := \underline{P \cdot (2^{2dr} \bmod M) \cdot 2^{-dr} \bmod M}$
3: **for** $i = n-1$ **to** 0 **do**
4: $C := \underline{C \cdot C \cdot 2^{-dr} \bmod M}$
5: **if** $E_i = 1$ **then**
6: $C := \underline{C \cdot P \cdot 2^{-dr} \bmod M}$
7: **end if**
8: **end for**
9: $C := \underline{1 \cdot C \cdot 2^{-dr} \bmod M}$

III. GPU と CUDA アーキテクチャ

CUDA[2], [3], [4] は NVIDIA 社が提供する C 言語の統合開発環境であり，NVIDIA 社の GPU に対してプログラミングモデルとインターフェイスを提供している．GPU high-level のハードウェアアーキテクチャを図 2 に示す．GPU は，大きく分けてプログラムを処理するユニットとデータを格納するメモリから構成される．演算処理ユニットとして，GPU は複数の Streaming Multiprosessor(SM) を持っている．各 SM 内の中には複数のコアが搭載されており，CUDA での処理はこれらのコアによって並列に実行される．また，CUDA のプログラミングモデルでは，図 3 に示すようにスレッド，ブロック，グリッドと呼ばれる階層構造を形成している．スレッドをまとめたものがブロック，ブロックをまとめたものがグリッドとなっており，プログラムやコンパイラはこれらを単位として取り扱う．それぞれのスレッドはブロック単位で各 SM に割り当てられ，並列に動作するスレッドを GPU 上の各コアに割り当てて処理を行う．SM では，連続した 32 個のスレッドをグループとして実行し，このグループのことをワープと呼ぶ．ワープ内の各スレッドは同一の命令を同時に実行するため，プログラマはワープ単位での操作を心がける必要がある．

CUDA には Global Memory と Shared Memory の 2 種類の共有メモリが存在する．Global Memory は大容量 (1.5-6GB) のオフチップメモリで，どのブロックに属するスレッドからでも読み書きが可能である．その一方で，メモリのアクセス速度は遅い．Global Memory 上では，同一ワープ内の各スレッドが離散したアドレスへアクセスする (stride access) と大きな遅延が生じるため，連続したアドレスへアクセス (coalesced access) するコアレッシングを用いることが重要となる．コアレッシングを用いると，ワープ内の全てのスレッドが連続したメモリ領域に同時にアクセスできるため，高速な処理が可能となる．Shared Memory は小容量 (16-48KB) のオンチップメモリで，メモリアクセス速度が速い．

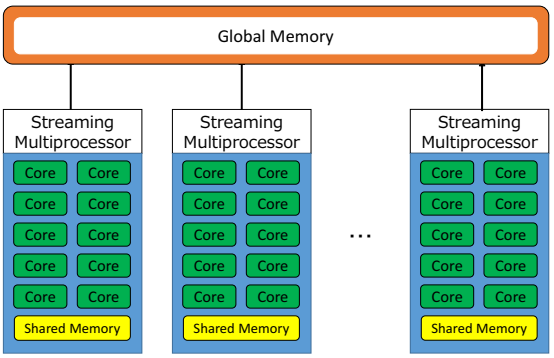


Fig. 2. GPU のハードウェアアーキテクチャ

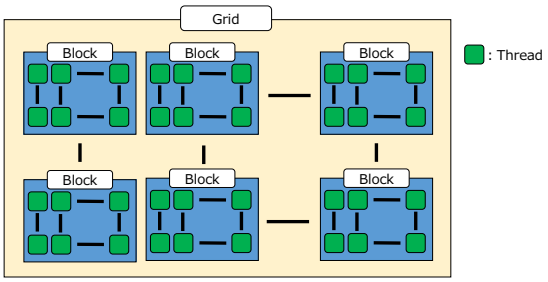


Fig. 3. CUDA のソフトウェアアーキテクチャ

Global Memory 上の二次元配列のデータへのアクセスについて考える (図 4). 各スレッドが水平方向のデータにアクセスする場合，アクセスしたいアドレスは連続している．このとき，各スレッドはコアレッシングを用いてデータを取得することができるため，高速にメモリアクセスを行うことができる．垂直方向のデータにアクセスする場合，リクエストしたアドレスはそれぞれ離れている．このとき，各スレッドはそれぞれのアドレスを逐次的に取得するため，メモリアクセスは低速となる．

IV. GPU 実装

本節では，RSA 暗号の GPU 実装について説明する．実装では効率的なメモリアクセスの実装，インラインアセンブラの利用の 2 つのアイデアを適用しており，それぞれについて説明する．実装は 1 つのスレッドに 1 つの平文の暗号化を割り当て，複数スレッドによって大量の平文を並列に暗号化していく．

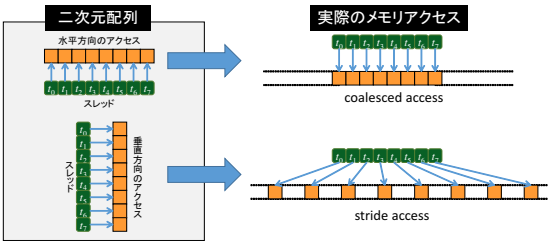


Fig. 4. メモリアクセス

A. 効率的なメモリアクセスの実装

モンゴメリ乗算の演算では，入力にかかわらず同じ配列の要素にアクセスするという特徴がある．与えられた平文 P をそのまま Global Memory に渡すと，アクセスしたいアドレスはそれぞれ離れたメモリ領域に格納されている．このため，各スレッドは分散したアドレスへアクセス (stride access) するために逐次的にデータを取得する必要がある．よって，大きな遅延が生じる (図 5)．これを解消するため，平文の入力データを Global Memory に渡す際に，各ブロックごとに転置を行う．このようにしてデータを渡すことで，アクセスしたいアドレスは連続したメモリ領域に格納される．このため，連続したメモリ領域に同時にアクセス (coalesced access) することができ，コアレッシングを実装したメモリアクセスを可能にする (図 6).

図 5 は，ワープ内のスレッドが i 番目の要素にアクセスする様子である．平文 P_0 の i 番目の要素は i 番目に格納されており，P_1 の i 番目の要素は $i+d$ 番目，P_2 の i 番目の要素は $i+2d$ 番目に格納されている．このように i 番目の要素は，Global Memory 上でそれぞれ離れているため，各スレッドはそれぞれのアドレスを逐次的に取得する．図 6 は，転置後の i 番目の要素にアクセスする様子である．平文 P_0 の i 番目の要素は di 番目に格納されており，P_1 の i 番目の要素は $di+1$ 番目，P_2 の 2 番目の要素は $di+2$ 番目に格納されている．このように i 番目の要素は連続して Global Memory に格納されているため，各スレッドはコアレッシングを用いてデータを取得できる．

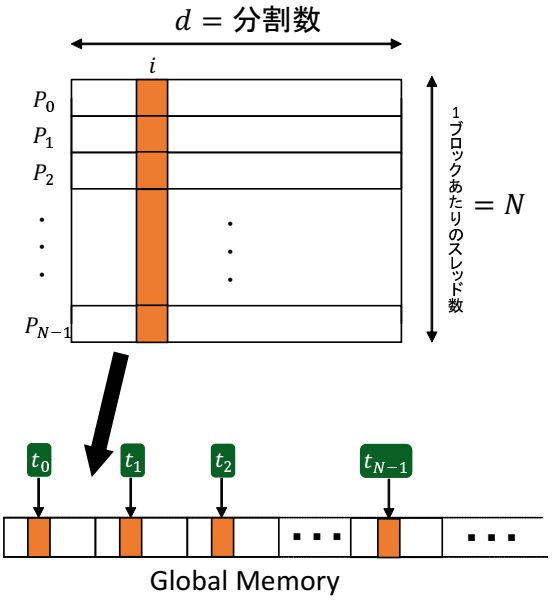


Fig. 5. 転置前のメモリアクセス

B. インラインアセンブラの利用

Algorithm 1 のモンゴメリ乗算では，積和演算を繰り返し行う．通常の実装では，32bit の積和演算はオーバーフローを起こす可能性があるため，64bit 整数に拡張して演算を行う．しかし，GPU では 64bit の積和演算は 32bit の積和

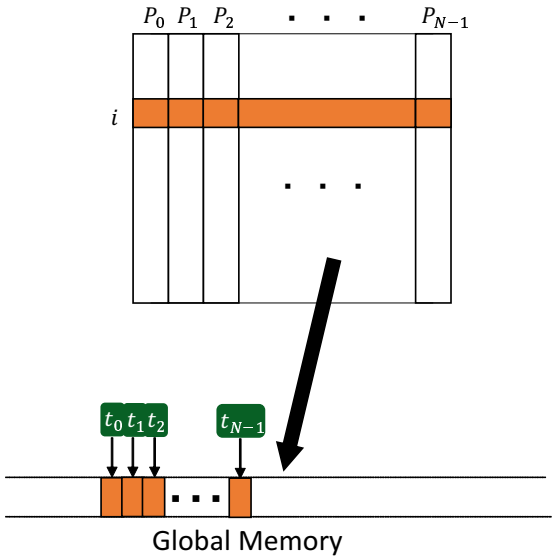


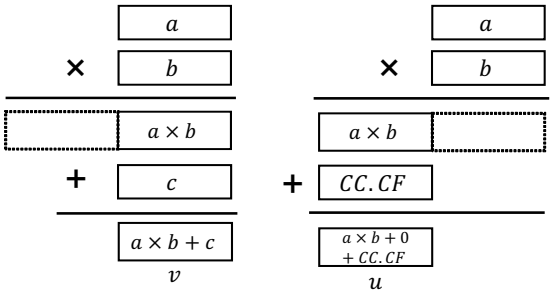
Fig. 6. 転置後のメモリアクセス

演算よりも時間がかかってしまう．そこで，インラインアセンブラを利用し，乗加算命令の”mad.cc”,”madc”を用いてアセンブリ言語で直接記述することにより高速化を行った [9]．”mad.cc”,”madc”はキャリーアウト格納付きの乗加算命令である．”mad.cc”はキャリーなしの積和演算で，”madc”はキャリー付きの積和演算を行う．モンゴメリ乗算で必要な積和演算は，加算が 1 変数 ($(u, v) = a \times b + c$) の場合と 2 変数 ($(u, v) = a \times b + c + d$) の場合の 2 種類がある．$u, v$ はそれぞれ積和演算の結果の上位 32bit，下位 32bit を表す．1 変数の加算の実装を図 7 に示す．$a \times b$ の下位 32bit の結果に c を加算する．この結果が v となる．このときキャリーアウトした場合は，レジスタ CC.CF に格納される．次に，$a \times b$ の上位 32bit の結果に CC.CF を加算する．この結果が u となる．このようにして 1 変数の加算の実装を行っている．2 変数の加算の実装を図 8 に示す．まず，$c + d$ を上位 32bit と下位 32bit に分割し，それぞれ x, y に格納する．次に，$a \times b$ の下位 32bit の結果に y を加算する．この結果が v となる．このときキャリーアウトした場合は，CC.CF に格納される．そして，$a \times b$ の上位 32bit の結果に x と CC.CF を加算する．この結果が u となる．このようにして 2 変数の加算の実装を行っている．

V. 実装結果と考察

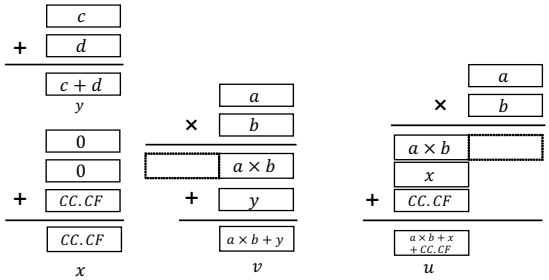
本研究では GPU に NVIDIA 社の GeForce GTX TITAN を利用した．GTX TITAN は 14 基の SM を持ち，コア数は 2688，グローバルメモリの容量は 6GB である．逐次処理と比較するため CPU で RSA 暗号を実装し，Intel 社の Xeon X7460 プロセッサ (動作周波数 2.66GHz) を単一コア/単一スレッドで実行した．

RSA 暗号は 1024bit と 2048bit で実装し，前節で説明した 2 つのアイデアを評価するために，メモリ転置の有無とインラインアセンブラの利用の有無の計 4 種類の実装を行い，実行時間の比較を行った．平文の数は 1048576 文である．平文は int 型に分割しており，分割数は 1024bit の暗号化では 32




```
asm("mad.lo.cc.u32 %0, %2, %3, %5;"
    "madc.hi.u32 %1, %2, %3, %4;"
    :"=r"(v),"=r"(u)
    :"r"(a),"r"(b),"r"(0),"r"(c));
```

Fig. 7. 1 変数の加算の積和演算




```
asm("add.cc.u32 %3, %6, %7;¥n¥t
 "addc.u32 %2, %8, %8;¥n¥t
 "mad.lo.cc.u32 %0, %4, %5, %3;¥n¥t"
 "madc.hi.u32 %1, %4, %5, %2;¥n¥t"
 :"=r"(v),"=r"(u),"+r"(x),"+r"(y)
 :"r"(a),"r"(b),"r"(c),"r"(d),"r"(0));
```

Fig. 8. 2 変数の加算の積和演算

個，2048bit の暗号化では 64 個となっている．メモリアクセスの比較では，与えられた平文をそのまま Global Memory に渡す手法と，転置を行うことでコアレッシングを実現したメモリアクセスの手法を比較する．このとき，インラインアセンブラはいずれの手法でも使用していない．また，転置にかかる時間は，1024bit の RSA 暗号化で 0.008[s]，2048bit の RSA 暗号化で 0.036[s] であり，実行時間に比べてかなり小さいため考えないこととする．次に，インラインアセンブラの利用についての比較では，64bit-int を用いた積和演算の手法と，インラインアセンブラを利用した積和演算の手法を比較する．このとき，いずれの実装もコアレッシングを実現した実装となっている．

実装は 1 つのスレッドに 1 つの平文の暗号化を割り当て，複数スレッドによって大量の平文を並列に暗号化していく．1 ブロックあたりのスレッド数によって，GPU 実装のパフォーマンスに影響を与える．よって，1 ブロックあたりのスレッド数 t を $t = 32, 64, 128, 256$ の 4 種類でそれぞれ実験を行

い，最も良い結果が出たスレッド数を選択する．CPU から GPU へのデータの転送時間は，実行時間と比較してかなり小さいため，考えないものとする．結果を表 I，表 II，表 III，表 IV に示す．

効率的なメモリアクセスの評価のために表 I，表 II を見ると，いずれの場合もコアレッシングを実現した実装の方が実行時間が速く，高速化を達成していることがわかる．1024bit の RSA 暗号では，$t = 128$ のとき 6.28 倍，2048bit の RSA 暗号では，$t = 256$ のとき 14.25 倍の高速化を達成している．インラインアセンブラの利用の評価のために，表 III，表 IV を見ると，いずれの場合もインラインアセンブラを利用した積和演算の手法の方が実行時間が速く，高速化を達成していることがわかる．1024bit の RSA 暗号では，$t = 32$ のとき 1.63 倍，2048bit の RSA 暗号では，$t = 128$ のとき 1.54 倍の高速化を達成している．

本研究で提案した 2 つのアイデアを適用した場合，GPU 実装と CPU の逐次処理と比較すると，1024bit の RSA 暗号では CPU 実装が 97.800[s] かかるのに対し，GPU 実装は $t = 32$ のとき 1.840[s] で暗号化でき，53.2 倍の高速化を達成している．2048bit の RSA 暗号では 1024bitRSA 暗号では CPU 実装が 368.975[s] かかるのに対し，GPU 実装は $t = 128$ のとき 6.539[s] で暗号化でき，56.4 倍の高速化を達成している．

TABLE I
1024BITRSA 暗号でのメモリアクセスによる実行時間 [s] の比較
(CPU:97.800[s])

t	stride access	高速化率	coalesced access	高速化率
32	16.840	5.808	2.995	32.656
64	16.851	5.804	2.691	36.346
128	16.874	5.796	2.687	36.402
256	21.242	4.604	3.031	32.267

TABLE II
2048BITRSA 暗号でのメモリアクセスによる実行時間 [s] の比較
(CPU:368.975[s])

t	stride access	高速化率	coalesced access	高速化率
32	103.658	3.560	14.137	26.100
64	143.671	2.568	10.070	36.642
128	147.207	2.506	10.068	36.649
256	143.192	2.577	10.048	36.721

TABLE III
1024BITRSA 暗号でのインラインアセンブラの利用による実行時間 [s] の比較 (CPU:97.800[s])

t	インラインアセンブラなし	高速化率	インラインアセンブラあり	高速化率
32	2.995	32.656	1.840	53.153
64	2.691	36.346	1.854	52.753
128	2.687	36.402	1.862	52.527
256	3.031	32.267	1.869	52.333

Jang らの実装 [5] との比較を表 V に示す．Jang ら実装では，多倍長整数の乗算は CPU 上で行い，依存関係のない処理を GPU 上で並列に実行している．本実装で使用している GPU は GTX TITAN であり，Jang らが使用している GPU は GTX 480 である．コア数を比較すると 5.6 倍である．スループットを見てみると，8.5 倍である．平文の長さや GPU の性能によって実行時間が大きく変化するため，直接比較することは難しいが，コア数の増加よりスループットの増加の方が大きいため，高速化に成功しているといえる．

TABLE IV
2048BITRSA 暗号でのインラインアセンブラの利用による実行時間 [s] の比較 (CPU:368.975[s])

t	インラインアセンブラなし	高速化率	インラインアセンブラあり	高速化率
32	14.137	26.100	9.410	39.210
64	10.070	36.642	6.545	56.371
128	10.068	36.642	6.539	56.427
256	10.048	36.721	7.107	51.919

TABLE V
1024BITRSA 暗号の既存手法との比較

	既存手法 [5]	本手法
GPU	GTX 480	GTX TITAN
コア数	480	2688
スループット [msg/s]	66970	569878

VI. まとめ

本研究では，RSA 暗号を GPU に実装し高速化を行った．効率的なメモリアクセスの実装，インラインアセンブラの利用により高速化を達成した．本研究の実装では，GPU 実装と CPU の逐次処理と比較すると，最大で 1024bit の暗号化で 53.2 倍，2048bit の暗号化で 56.4 倍の高速化という結果を得られた．

REFERENCES

[1] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, Vol. 21, No. 2, pages 120–126, 1978.
[2] NVIDIA Corporation. CUDA C Programming Guide Version 5.5, 2013.
[3] NVIDIA Corporation. CUDA C Best Practice Guide Version 5.5, 2013.
[4] NVIDIA Corporation. CUDA ZONE, http://developer.nvidia.com/category/zone/cuda-zone.
[5] K. Jang, S. Han, S. Han, S. Moon, and K. Park. Accelerating SSL with GPUs. *ACM SIGCOMM Computer Communication Review*, Vol. 41, issue 1, 2011.
[6] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, Vol. 44, pages 519–521, 1985.
[7] Ç. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and Comparing Montgomery Algorithms. *IEEE Micro*, Vol 16, No. 3, pages 26–33, 1996.
[8] S. Neves, F. Araujo. On the Performance of GPU Public-Key Cryptography. *IEEE International Conference on Application-Specific Systems, Architectures and Processors(ASAP)*, 2011.
[9] NVIDIA Corporation. Parallel Thread Execution ISA Version 3.2, 2013.
[10] S. Bo, K. Kawakami, K. Nakano, and Y. Ito. An RSA Encryption Hardware Algorithm using a Single DSP Block and a Single Block RAM on the FPGA. *International Journal of Networking and Computing*, Vol. 1, No. 2, pages 277–289, 2011.